January 2015

Refining Transformation Rules For Converting UML Operations To Z Schema

Tamaike Brown

REFINING TRANSFORMATION RULES FOR CONVERTING UML OPERATIONS TO Z SCHEMA

by

Tamaike M. Brown
Bachelor of Science, University of Technology Jamaica, 2011

A Thesis

Submitted to the Graduate Faculty

of the

University of North Dakota

in partial fulfillment of the requirements

for the degree of

Master of Science

Grand Forks, North Dakota
August
2015

This thesis, submitted by Tamaike M. Brown in partial fulfillment of the requirements for the Degree of Master of Science from the University of North Dakota, has been read by the Faculty Advisory Committee under whom the work has been done and is hereby approved.

Dr. Emanuel Grant, Chair

Dr. Andre Kehn

Thomas Stokke

This thesis is being submitted by the appointed advisory committee as having met all the requirements of the School of Graduate Studies at the University of North Dakota, and is hereby approved.

Dr. Wayne Swisher,
Dean of the School of Graduate Studies

July 20, 2015
Date

PERMISSION

Title	Refining Transformation Rules for Converting UML Operations to Z Schema
Department	Computer Science
Degree	Master of Science

In presenting this thesis in partial fulfillment of the requirements for a graduate degree from the University of North Dakota, I agree that the library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by the professor who supervised my thesis work or, in his absence, by the chairperson of the department or the dean of the Graduate School. It is understood that any copying or publication or other use of this thesis or part thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of North Dakota in any scholarly use which may be made of any material in my thesis.

Tamaike Brown

July 17, 2015

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisory committee for expressing interest and agreeing to be a part of my research endeavor. I would also like to thank God for giving me strength and the will of perseverance to reach yet another pinnacle in my life. A special thank you to Dr. Emanuel S. Grant whose vision inspired this work and whose guidance and expertise were crucial to the completion of my work. Thanks to Dr. Andre Kehn and Thomas Stokke as their insightful viewpoints and discerning revisions enhanced my work.

And last but not least to family and friends for their continued support and words of encouragement.

ABSTRACT

The UML (Unified Modeling Language) has its origin in mainstream software engineering and is often used informally by software designers. One of the limitations of UML is the lack of precision in its semantics, which makes its application to safety critical systems unsuitable. A safety critical system is one in which any loss or misinterpretation of data could lead to injury, loss of human lives and/or property. Safety Critical systems are usually specified by very precisely and frequently required formal verification. With the continuous use of UML in the software industry, there is a need to augment the informality of software models produced to remove ambiguity and inconsistency in models for verification and validation. To overcome this well-known limitation of UML, formal specification techniques (FSTs), which are mathematically tractable, are often used to represent these models.

Formal methods are mathematical techniques that allow software developers to produce softwares that address issues of ambiguity and error in complex and safety critical systems. By building a mathematically rigorous model of a complex system, it is possible to verify the system's properties in a more thorough fashion than empirical testing.

In this research, the author refines transformation rules for aspects of an informally defined design in UML to one that is verifiable, i.e. a formal specification notation. The specification language that is used is the Z Notation. The rules are applied to UML class diagram operation signatures iteratively, to derive Z schema representation of the operation signatures. Z representation may then be analyzed to detect flaws and determine where there is need to be more

precise in defining the operation signatures. This work is an extension of previous research that lack sufficient detail for it to be taken to the next phase, towards the implementation of a tool for semi-automated transformation.

CHAPTER I
INTRODUCTION
1.1 Research Definition

The motivation for doing this work stems from research that have been conducted previously at the University of North Dakota on unmanned aerial vehicles (UAVs). Clachar *et. al.* conducted research on Formalizing UML software models of safety critical systems [1]. The researchers used formal specification to enhance the imprecision semantics of UML and analyze its significance to safety critical systems. As a result, a systematic process to transform ambiguous UML class diagrams to a formal representation for verification and validation was devised. In a follow up study conducted by Jackson *et. al.* the researchers conducted a case study on validation and verification of object-oriented design using formal specification technique [2]. In that study a preliminary set of rules were defined for transforming UML class diagrams methods to formal specification (Z notation) [2]. The transformational rules they developed were not complete and were at a very high level of abstraction. In order for these transformation rules to be productive, they necessitates further elaboration and refinement. Hence, this research attempts to produce a complete set of transformation rule for converting UML class operation signature to Z notation/description.

1.2 Benefits of the Research

Unmanned Aircraft Systems (UASs) have been in existence for many years. UASs commonly referred to as Unmanned Aerial Systems is defined as a system, whose components include the air vehicles and corresponding hardware that do not involve a human operator, but instead maneuver autonomously or are remotely piloted. UAS must be considered in a system context, which encompasses the command, control and communications systems, and personnel necessary to control the unmanned aircraft [3] [61]. However, recently, the use of UASs has experienced immense growth and UASs play a central role in scientific research, defense, and in certain industries [3] [4]. In recent past, the use of UASs technologies lie at the core of military operations such as spacecrafts, aircrafts, helicopters, free-flying robots or mobile robots, surveillance, target identification and designation, mine detection, and reconnaissance [3] [4]. As their use continues to evolve, research has peaked on this technology to discover its applicability to other domains. UAS technologies are categorized as safety critical systems. This is due to them being utilized in high-risk tasks that require thorough development methodologies to guarantee their integrity. Today, UASs are involved in high-risk tasks such as border and port surveillance by the Department of Homeland Security, assist with scientific research and environmental monitoring by National Aeronautics and Space Administration (NASA) and National Oceanic and Atmospheric Administration (NOAA [54]. A system that is defined as safety critical can have serious ramifications if an error occurs. These implications include the risk of injury, loss of life, data, and property. Therefore, designing these systems requires: 1) thorough understanding of their requirements, 2) precise and unambiguous specifications, and 3) metrics to verify and validate the quality of software produced.

In order for safety critical aviation systems to be accepted by the Federal Aviation Administration (FAA) and other interested parties, they must adhere to standards such as the RTCA DO-178B and its current descendant D0-178C [5]. The DO-178B is an airworthiness compliance standard, which governs the development and certification of aerospace systems. It is a process-oriented evaluation of sound software engineering practices in system design [6]. The standard focuses on all aspects of round trip engineering and requirements based testing as key elements of software verification to uncover errors [6]. DO-178C is the latest revision to the DO-178B guidelines, which addressed objectives for software development life cycle processes, activities and design considerations for achieving the objects and verifying the objectives [50]. DO-178C also addresses object-oriented development concepts and specific techniques.

The University of North Dakota (UND) – UAS Risk Mitigation Project[1] was awarded a contract to develop a proof-of-concept air truth system, which monitors the operation of UAVs in the US National Airspace (NAS). The project began with minimal requirements; however, the timeframe for delivery was very rigid. This resulted in the rapid development of a prototype to assist in exploring and developing additional requirements. One feature of prototypes is that they are often poorly documented. To resolve this, concurrent definition and documentation of system requirements were performed as the prototype evolved. This was enhanced with the design of graphical software models. In model-driven engineering, the purposes and uses of graphical software models are multifaceted. They represent the structural design of the system, and the flow of data and communication between the various systems and subsystems. Its use is not only suited for astute stakeholders but also non-technical stakeholders such as customers – to convey how their requirements are being met. The Unified Modeling Language (UML) is an ISO standard for

[1] http://www.uasresearch.com/home.aspx

designing and conceptualizing graphical models of software systems [7]. Since its development by the Object Management Group (OMG)[2] in the early 1990's its use has increased in industry and academia. Graphical software models, such as UML models, possess simplistic designs and promote good software engineering practices. However, these models are not without limitation. Graphical software models are often imprecise and ambiguous. In addition, they are not directly analyzable by type checkers and proof tools. This makes it difficult to evaluate the integrity and correctness of its models; therefore, valid assertions cannot be made with regard to meeting user requirements.

Formal Specification Techniques have been advocated as a supplementary approach to amend the informality of graphical software models [8] [9]. They promote the design of mathematically tractable systems through critical thinking and scientific reasoning. FSTs use a specification language, for instance Z notation, to describe the components of a system and their constraints [10]. Unlike graphical models, formal models can be analyzed directly by a proof tool – which checks for errors and inconsistencies. Critics of FSTs claim, they increase the cost of development, require highly trained experts, and are not used in real systems [11]. Yet, they have been used in case studies which unveiled that, FSTs facilitate a greater understanding of the requirements and their feasibility [1] [2] [12]. Although the use of FSTs is sometimes controversial, their benefits to critical systems offset the disadvantages.

This work documents the transformation rules for UML class operation signature to an analyzable representation using formal specification techniques. Equally, the specific advancement that this work encourages is to provide a mean by which these transformation rules can be automated. Automation is necessary because of the volume involve in such work – manual

[2] www.omg.org

interventions can be monotonous and inaccurate. Such process will reduce the introduction of human errors when applying transformation rules.

1.3 Research Contribution

Previous research has demonstrated that the application of formal specifications to safety critical systems is important for the purpose of precision. The present work is designed to define a set of rules for transposing UML operations to Z schema, which is an extension of work done by France *et. al*. [8] [12] [16] [24]. This research introduces four steps that are applicable to any domain that is categorized as safety-critical and where formalism is necessary. The present analysis demonstrates that it is feasible to apply formal specifications to safety-critical systems, although the manual process is tedious and the use of notations are necessary. The present research is therefore intended to make contributions to the literatures on formal specifications, and UML models. In addition, this work may lead to the production of a similar tool highlighted in work done by Gogolla *et. al.* for UML and OCL validation [62].

1.4 Research Approach

UML is now an ISO standard [7] and has its advantages in simplicity, intuitiveness and recently has been considered for specification purposes. However, UML falls short in the latter area because it utilizes some loose semantics, which leads to ambiguity among its models. In some cases, ambiguity can be negligible, however in safety critical systems this may lead to detrimental consequences. One technique to eliminate this ambiguity is by transforming UML models to an analyzable representation with the use of formal specification techniques. Prior work has been conducted in formalizing UML class diagram operation signatures at an abstract level [2]; from that research, it requires that in order for those transformation rules to be effective, it demands elaboration and refinement. This effort will look at how UML class diagram operation signatures

can be formalized by applying rules to user-defined functions using a formal specification language, Z notation.

According to [1], formal specification techniques (FST) incorporate the use of a specification language to describe software models with precision. As noted previously, the specification language utilized for our research is Z. FST also permits the use of proofing tools which identify errors in specifications executed within the proofing tool environment. The employment of FST will look at checking and analyzing the Z schemas that have been yielded from the system's UML class diagram. A proof tool used to accomplish this which, has shown to be effective in detecting syntax and semantic errors of the Z representation of our UML model is Z/EVES. Carrying out a series of analysis of error checking using this proofing tool is a key element in the validating system models.

In an effort to automate model transformation in the future as a byproduct of this research, a set of model transformation rules will be highlighted throughout the methodology. Model transformation works by accepting one or more models, by applying rules called transformation rules, a target model is then attained which is equivalent to the input model [1]. Transformation is currently being conducted manually however, with the establishment of detailed transformation rules the process can be done automatically. As a byproduct of this research, automatic UML model transformation into their equivalent Z schemas will be a focus in future works. To aid in this potential research, the methodology aims to highlight a set of transformation rules, which will be used to accomplish automatic model transformation.

EBNF (Extended Backus Normal Form or Backus–Naur Form) is a recursive notation technique for describing the productions of a context-free grammar. It is developed based on the work of John Backus with contributions by Peter Naur [17] [18]. It is often used to describe the

syntax of languages used in computing, including computer programming languages, because of its simple notations, recursive structures, and it is widely supported by many compiler generations tools such as YAAC [19], LEX [20], and ANTLR[21]. BNF is applied wherever exact descriptions of languages are needed for instance, in official language specifications, in manuals, and in textbooks on programming language theory. It is realized in applications that the descriptive power of BNF may be greatly improved by introducing a few extended meta-symbols, particularly those for repetitive and optional structures of grammar rules.

Extended Backus Naur Form (EBNF) will be used to describe UML operation signature. After which, the researcher will go through a process of systematically deriving an algorithm definition for each transformation rule – the refinement process. Having refined the rules, testing of each algorithmic description will be done on a case study. From the case study, conclusion will be drawn of the result as it relates towards determining the success of the effort.

1.5 Scope and Limitations

The effort of this work is limited to defining a set of algorithms for transforming UML operation signature that are presented in a complete UML class diagram. This research draws on work done by [1] [2] and will not attempt to redefine any rules define by [1], but merely drawing on the stated output of the researcher work [1]. In addition, building of an application will not be an attempt in this study but defining a set of algorithms that could be implemented in such an application.

Limitations encountered relate to timing constraints and available work effort. The process of deriving formal rules is very lengthy hence, the manual efforts involved in developing them spanned several semesters. The breakdown is as follow:

Semester 1 – Literature review as an independent topic

Semester 2 – Deriving rules

Semester 3 – Case study

Semester 4 – Case study

Semester 5 – Case study

Transformation rules and case study were done in two semesters iteratively and another semester was spent writing the thesis.

1.5.1 Expected Outcome

The output of this reports include: 1) a set of rules and algorithms for transforming UML operations to Z schema. 2) Results from the case study and a conclusion of the success of the work and its future implementation. The algorithms will constitute the high-level pseudo-code description that would lead to implement a system that will conduct the transformation in a semi-automatic manner. The transformation cannot be fully automated as some operations constraints can only be specified in English prose.

1.6 Description of Thesis / Report Organization

Chapter 2 incorporates areas that were reviewed for doing this work as well as prior research in this sub-discipline. Chapter 3 contains a detailed description of the methodologies performed in this research. Chapter 4 is a case study on the UAS and results of the transformation rules to the system. The report is concluded in Chapter 5.

CHAPTER II
BACKGROUND
2.1 Model Driven Approach

The focus of Model Driven Engineering (MDE) is to transform, refine, and integrate models into the software development life cycle to support system design, evolution, and maintenance [22]. Models serve many purposes and their use varies from investors to investors. The purpose of modeling, from a developer's standpoint, is to represent the proposed system by showing: 1) the flow of data between objects and individual components of the system as well as how they can interact with other software components, 2) Communication between internal entities and external components, and 3) how the system behaves to stimuli.

Models should be logical, cohesive, and provide an abstract way to visualize the design of a system and show how the proposed system will address the users' requirements. One way in which models can be derived is by forward engineering activities. Forward engineering is the process of moving from high-level abstractions and implementation of independent designs to the implementation of a system [23].

Software models facilitate:

- Abstraction - This feature allows models to be independent of any programming language, style, or algorithm design.
- Improved understanding of the project's goals and user's requirements - Members of the development team(s) can distinctly see how the proposed solution addresses the customer's needs, and the impact that their aspect of work has on meeting these requirements.

- Enhanced communication between the various stakeholders. The intent of some models is to be conveyed to all stakeholders. The outcome is to encourage all stakeholders to play an active role in the design of the system and to explore the feasibility of its requirements.

Various software development life cycle models are suitable for specific project related conditions, which include organization, requirements stability, risks, budget, and duration of project. One life cycle model theoretical may suite particular conditions and at the same time another model may also looks fitting into the requirements but one should consider trade-offs while deciding which model to choose.

While Software models have many benefits, their disadvantages include the following:

- On occasion, models are not updated which results in them becoming inconsistent with the source code. This affects the maintainability of the software as the models are not a true representation of their implementation.
- Graphical models are abstract hence the software developer is not required to explore certain aspects of the system; for instance side effects related to variable declarations or premature initializations.
- Failure to detect syntax, semantic and domain errors because these models cannot be directly verified for inconsistencies without the use of an external tool.

Even though the use of software models can be unfavorable, it is still an essential step in the design, documentation, and maintenance of software systems. The result of modeling determines whether the models are indicative of the proposed system and if the user's needs are adequately addressed. There are many types of models and an excess amount of software tools used to aid in their design. This research uses the UML to support the design of graphical models. UML is appropriate because:

- It is an ISO Standard for designing models of software systems,
- It is widely used in industry and academia, and
- It is user friendly and understandable for all stakeholders.

2.2 The Unified Modeling Language

UML (Unified Modeling Language) is the de-facto standard formalism for object-oriented software analysis and design. One of the most consequential components of UML are class diagrams, which model the information on the domain of interest in terms of objects organized in classes and relationships [40] [41]. UML class diagrams allow for modeling, in a declarative way, the static structure of an application domain, in terms of concepts and relations between them.

A class in a UML class diagram represents an object or a set of objects with common features. A class is graphically rendered as a rectangle divided into three parts (see Figure. 1). The first part contains the name of the class, which has to be unique in the entire diagram. The second part contains the attributes of the class, each denoted by a name, possibly followed by the multiplicity and with an associated type for the attribute values [42]. The third part contains the operations of the class, that is, the operations associated to the objects of the class. The argument list is a list of parameter types (e.g., int, double, string, etc.) that is associated with operation. Operations that does not return a value should give a return type of void.

The UML is an object-oriented modeling language for specifying, visualizing, constructing, and documenting the artifacts of software systems [7]. The UML is used to depict a high-level representation of the proposed system. This is achieved through the design of various types of models, which capture the structure and behavior of the system. UML promotes some of the best software development practices; and this very quality is among the primary reasons for its acceptance. It serves as a blueprint for software engineers through the design of models and

diagrams, which are representative of various aspects of the proposed system. The benefits of UMLs are very present in the early phase of the software development life cycle where it is used to reproduce a high-level representation of the proposed system. This abstract representation is achieved through the design of various types of models, which capture the structure and behavior of the system, sub-systems, and their internal and external components.

UML models facilitate better communication among customers and developers. Customer and developer get an opportunity to understand the project and its requirements before implementation commences. It also assists software developers to identify whether user requirements will be adequately addressed by the system. UML is widely accepted because of its simplicity, which makes it easily understood by developers thereby making it easily communicated to their customers [26].

Diagrams in UML are categorized as structure or behavior diagrams. Structure diagrams represent the static framework of the system [27], whereas behavior diagrams illustrate the dynamic features of the system. Examples of structure diagrams include class, component, object, deployment, and package diagrams. Behavior diagrams depict the dynamic features of the system by showing how the system act during execution. These diagrams include use case, activity, and state diagrams. Interaction diagrams are an extension of behavior diagrams but focus mainly on the internal elements of the system. Examples of interaction diagrams include sequence and collaboration diagrams. Class diagrams and use case diagrams facilitate prioritization and communication between nontechnical stakeholders and developers. Sequence and state chart diagrams that are more complex UML models, and are suited for intellectual advance stakeholders such as engineers and developers.

The scope of this paper will be on the static UML models – more specifically, the class diagram. Creation of a new class diagram in UML begins with a class. In UML notation, a class is represented as a rectangular box with three vertical compartments: the class header, list of attributes, and list of operations. Attributes are characteristics of a class that makes it unique; whereas operations, also called methods, performs tasks that could potentially change the state of the class. The focus of this work is on the class operations.

Figure 1 below illustrates an example of two classes that have some mutual relationship depicted by the line connecting them. Each has its own unique attributes (for example, Class A has Attribute_A0 through to Attribute_An) and some operations listed below the attribute list. Note that operation signatures may contain a list of parameters as well. For the scope of this paper, focus will be placed on the operation signatures of classes. Previous research work done by Clachar *et al.* [1] focused on defining classes and attributes; and the need for greater attention on operation signatures should be done in order to have a complete transformation model to Z.

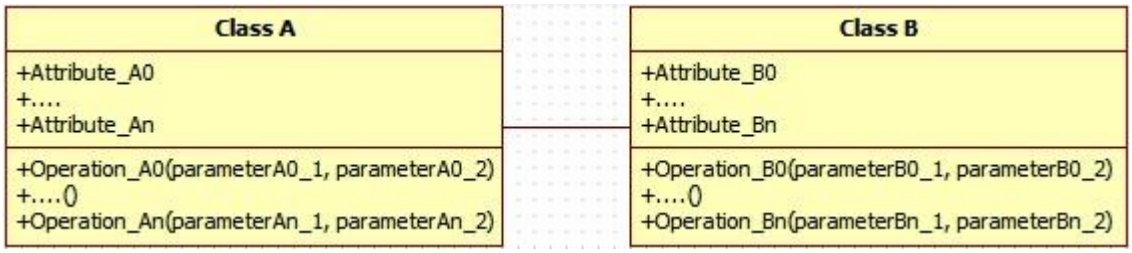

Figure 1. Example of a UML class diagram

Like other software development aids, as recorded before UML has its limitations. These informal models have an advantage, such as expressiveness – which makes the objective of the system easily conveyed to both technical and nontechnical stakeholders. However, UML lacks precise formal semantics, which results in its models being subject to multiple interpretations. This issue is worsened by the use of natural language annotations – as a means of clarification and

explanation of the modeling techniques adopted. Due to UML's inherent flexibility, developers are given much scope when designing models. This freedom enables the developer to describe system requirements based on the modeling technique they have adopted. However, problems arise when these models are circulated among the development team and each developer interprets the models in a different way – which could affect the latter stages of the software development life cycle (SDLC) [28]. Notations are often used to alleviate this issue; however, comments can be misinterpreted because it is expressed in natural language [16]. Furthermore, natural language notations cannot be processed by tools – therefore they also need to be formalized for analysis purposes [29]. Other problems arise as customer requirements unfold. These critical changes are often not reflected in the models – albeit the source-code reflects the change; at that stage updating the models is often considered tedious and time-consuming. This result in difficulty of software maintenance as the UML models are often inconsistent with the source code and its significance is lost [30].

In some systems, the disadvantages of UML and the challenge of deriving precise models may not have a significant impact on the quality of software produced. Yet, in safety critical systems, any inadequacy could result in the loss of property or be life threatening. The high cost during the implementation and early test phases are often times caused by errors at the specification and design phases [25]. Since UML is widely accepted, there is a need for methods to test the correctness of its models. This can be achieved with the use of formal specification techniques.

2.3 Formal Specification Techniques

Formal specification has been in existence decades afore the inception of graphical techniques such as UML. FSTs utilize mathematical models and principles to describe software models with accuracy through rigorous analysis [3]. The specification language chosen in this

work is referred to as Z notation for the following reasons. (1) It is an ISO standard, (2) There is an excess of research on Z notation and extensive implement support (3) It is based on set theory and predicate logic, which allows mathematical reasoning by categorizing real world entities into sets (4) Similarity with constructs used in UML – thereby making the transformation process easier to grasp.

Developed at Oxford University, Z is a typed language based on set theory and first order predicate logic. As well as a basic mathematical notation, Z includes a schema notation to aid the structuring of specifications [13]. In order to develop schemas, Z language uses typed mathematical facts including sets, relations, and functions in conjunction with first order predicate logic. Schemas define its relevant variables and specify the relationship between the values of the variables. A schema describes the stored data that a system accesses (variables) and alters [14]. A basic type is like a typical data type such as integer, natural number etc. however; it is user-defined and problem specific. A schema may include one or several basic types.

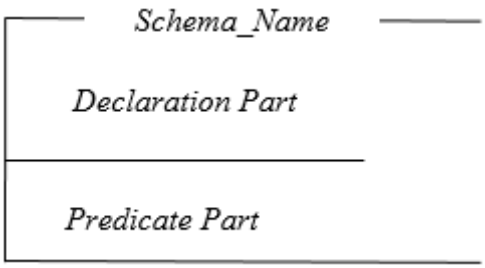

Figure 2. Structure of a Z Schema

[*Basic_Type*]

Figure 3. A Basic Type Representation in Z

There are two [2] representation of schemas: state schemas and operation schemas. State schemas are employed to define the static attributes of a system while operation schemas capture

dynamic aspects [16]. For the purpose of this research focus is placed on defining rules for user-defined functions by formalizing UML class diagram, the operation schema is the schema of focus to demonstrate our methodology.

A specification written in Z notation models the proposed system by specifying the components of the system and expressing constraints between those components [10]. Because of its formal basis it enables mathematical reasoning, and hence proves that, desired properties are outcomes of the specification [10]. From these proofs, one can verbalize that the system is behaving in a desirable or undesirable manner, provided the specification is precise and complete. System behavior should always be deterministic (deterministic in the sense that all events has a specified system response) in the domain of safety critical systems. These software systems encompass numerous highly intricate processing components and have high demands for reliability and accuracy. Due to the perpetual utilization of UML in software development, there is a need to resolve the informal semantics of the models it produces [1]. To transform UML models into Z notation also provide formal analysis to accomplish verification and validation of software systems.

Unlike UML, the formal models produced by Z can be analyzed directly by a proof tool – which checks for inaccuracy and inconsistencies. Possible errors that are detected include syntax and type errors, and domain checks – such as division by zero [31]. Inconsistencies that are detected pertain to the meanings of predefined and user-defined expressions and the appropriateness of their use in a specification. FSTs are not utilized to replace graphical software models; rather they are complementary. While formal models uncover inconsistencies and exclusion of requirements, the informal model is an explainable version of the formal models [24].

2.4 Model Transformation

Model transformation and refinement is a process that lies at the heart of model driven engineering (MDE), where platform independent models (PIM) are translated into platform specific models (PSM) utilizing formal rules – additionally referred to as transformation rules [32] [33]. The focus of MDE is to create and exploit domain models (that is, transform, refine, and integrate models), which are conceptual models into the software development life cycle to fortify system design, evolution and maintenance [22] [32]. The benefits of MDE was recognized and embraced by many organizations, including the Object Management Group (OMG) [22] – an association that creates and manages industry standards such as the UML. There are many categories of model transformations that exist such as text-to-model transformation, model-to-code transformation, and model-to-model transformation [22]. Although this work fixates on the latter, it will however also highlight the process of deriving the platform independent models. The platform independent models will be the UML class diagrams and the platform specific models will be their representative Z schemata.

After the models are transformed, theoretical properties of the transformation such as termination, soundness, completeness and correctness can be proven [22]. Irrespective of the transformation approach taken, it is vital that software engineers have a good understanding of the scope of the project, as well as the abstract syntax and semantics of the source and target models [32]. Models can be transformed manually or automatically. A manual transformation applies custom transformation rules to specific problems. This type of transformation was employed in this work. Whereas automatic transformations apply predefined transformation rules that are based on a problem domain. These rules can also be regarded as a meta-model.

This research seeks to derive a set of manual transformation rules for a real world unmanned aerial system that are applicable to all problem domains. The outcome of this activity is to determine if there are standard processes for yielding formal models from informal UML models for the problem domain. Manually transforming these models is tedious and as such, it is prone to human errors. Consequently, if standard processes were established, it would prove advantageous to automate them in future work. Conducting a manual transformation will highlight patterns that suit automation and aspects that require indispensable human intervention.

2.5 Extended Backus Naur Form

The extended Bankus-Naur form is the most rigorous way to define syntax of programming languages. EBNF is a notation for formally describing syntax. That is, how to write entries in a language [37] [38]. The use of EBNF will be used throughout this study to describe syntax formally. However, there is a more compelling reason to begin the use of EBNF: it is a microcosm of programming itself. In [39] Yong Xia and Martin Glinz have proposed a mapping from graphical language to EBNF aiming at the elimination of inconsistencies and ambiguities in UML diagrams. Complicated EBNF descriptions are easier to read and understand if their rules are well named, each name helps to communicate the meaning of its definition. However, to a compiler, names cannot change the meaning of a rule or the classification of a symbol [37].

Although, not incorporated in this work, but for future work EBNF can be used to describe C syntax formally. First, the control forms in EBNF rules are strongly similar to the basic control structures in C: sequence; decision, repetition, and recursion; also similar is the ability to name descriptions and reuse these names to build structures that are more complex. There is also a strong similarity between the process of writing descriptions in EBNF and writing programs in C: we must synthesize a candidate solution and then analyze it - to determine whether it is correct and

easy to understand. In this study, EBNF will be used to define syntax as a textual meta-model for the operation signature.

2.6 Related Tools – Z/EVES

There are many communities that are involve in developing a set of tools for editing, type checking, animating, and proving formal specification written in the Z specification language. However, many of them are command line tools and accept specifications in the Z Latex style [34]. Other implementations such as Z/EVES, CZT: Community Z Tools Project and RoZ have graphical interfaces that enable users to create Z specifications in a more user-friendly environment, while ensuring strict correspondence between the UML model and the Z schemata [35]. For the purpose of this study Z/EVES will be used to demonstrate the application of the define methodology.

Z/EVES offers some powerful automatic commands for general simple theorems proving for Z notation (e.g., prove or reduce) [51] [52]; it also has the ability to demonstrate the consistency of specification or refinements.

CHAPTER III
METHODOLOGY
3.1 Introduction

This chapter contains a detailed description of the methodologies performed in our research. The methodology involves three processes: 1) Using the operation signature description in EBNF. The rationale for using EBNF is that it provides a meta-model that it is easier to convert from textual (UML) to another textual format (EBNF). 2) Defining operation transformation rules, and 3) Converting transformation rules to algorithms.

This research is based on efforts of previous work [1] [2]. These work, focused on formalizing UML software models of safety critical systems, and validating and verifying functional design for complex safety critical systems. In addition, rules for transforming UML graphical models to Z notation were defined. This research completes the transformation rule by defining a set of rules that must be followed for defining operations in a class. Figure 4 shows an example of a class diagram base on the Unmanned Aircraft Systems (UAS) [1].

What follows in this research is a description of a series of sequential steps that will be carried out in transforming UML operations to Z representation. As each step is defined, it will be demonstrated by applying the rules to the operations shown in Figure 4. The operations that the rules will be applied to are:

- *convert_to_internal_speed (speed: Double): Double*
- *convert_to_external_speed (speed: Double): Double*
- *convert_heading (heading: Double): Double*

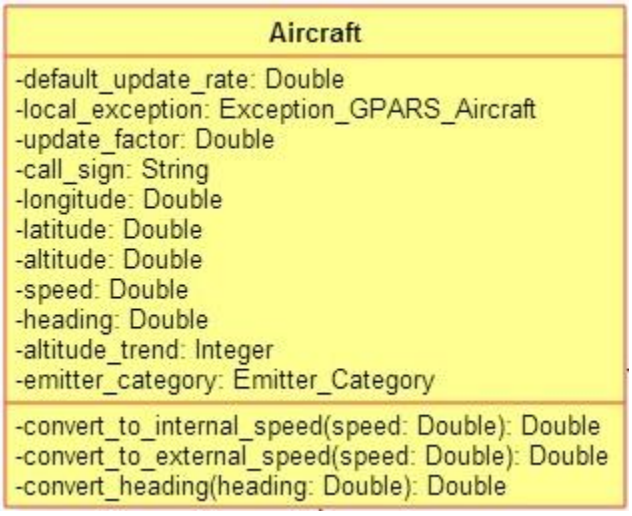

Figure 4. Example of a UAS Class Diagram

3.2 Extended Backus–Naur Form (BNF) Parsing rules for operations signature transformation

Extended Backus Normal Form (EBNF), a syntactic meta-language, is a notation technique for expressing context-free grammars in computer science. It is often used where clear formal description and definition is required to describe the syntax of languages used in computing, including computer programming languages [55]. EBNF is applied wherever exact descriptions of languages are needed, for instance, in official language specifications, in manuals, and in textbooks on programming language theory [37] [38]. In this work, EBNF will be used to define a set of formats for operation signatures. Any develop automated tool will have to implement the EBNF operation signature format so that operation can be parsed to production rule.

While the UML model represents operation signature as textual, our work would be more understandable if a textual meta-model representation is utilize. The scope of this meta-model is a class diagram. The textual description that is appropriate for this work is EBNF. Adhering to the ISO/IEC 14977: 1996 international standard, Figure 5 illustrates a meta-model class diagram description for a class operation signature.

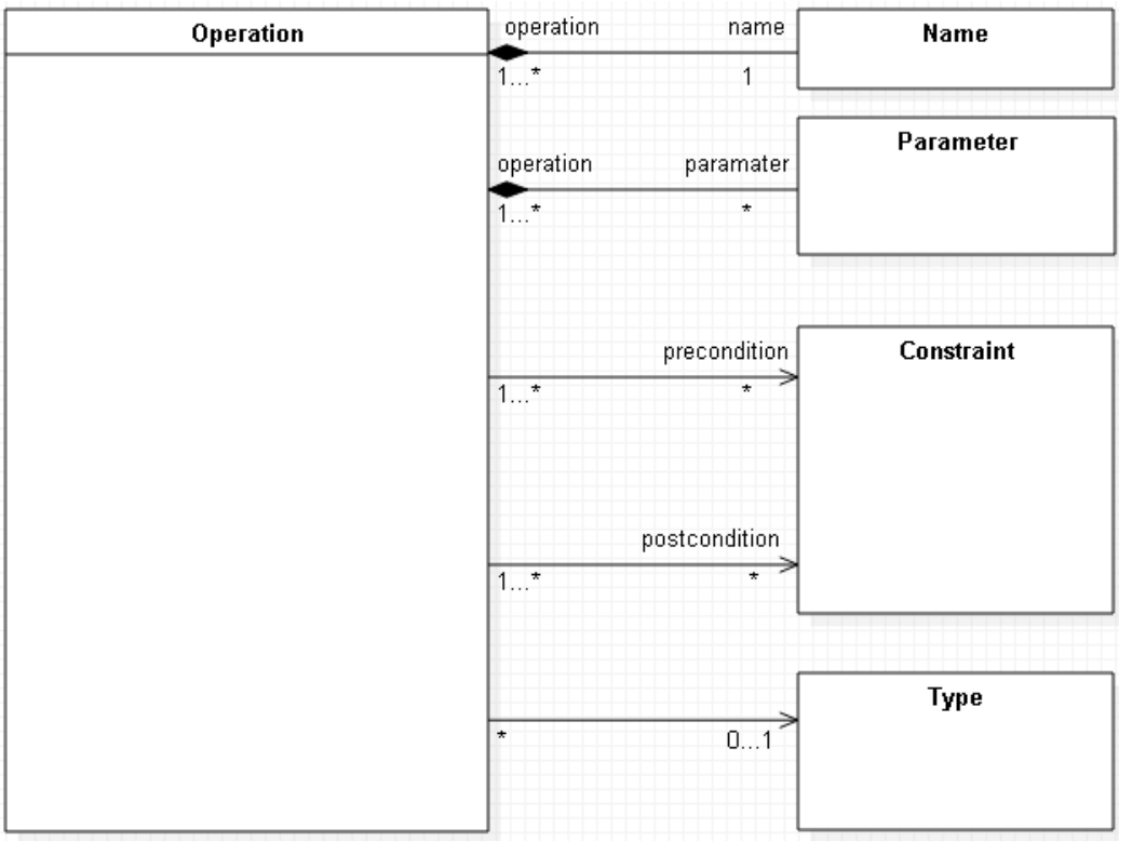

Figure 5. Meta-model description of a Class Diagram

For the purpose of this study, constraints that govern operations names are:

1. Size of operation name will follow the C standard 5.4.2.1 translation limits. 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character and 31 significant initial characters in an external identifier [56].
2. Words that conflict with key word in UML and Z should not be used.

The following EBNF grammar rules shown in Table 1 are adhered to when converting UML operations to EBNF:

Table 1. Rules for Converting UML Operations to EBNF.

<operation_signature> :: = <return_type><operation_name>“(“<parameters>”)” <constraint>
<return_type> :: = <z_type> |<user_defined_type>
<z_type> :: = $\mathbb{Z}$|$\mathbb{N}$
<user_defined_type> :: = void | char | string | short | long | float | double | signed | unsigned | char_string
<char_string> :: = <letter><more_letter>
<letter> :: = <upper_letter>|<lower_letter>
<upper_letter> :: = A | B | C | D | E | F| G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<lower_letter> :: = a | b | c | d | e | f | g | h | I | j | k | i | m | n| o | p| q | r | s | t | u | v | w | x | y| z
<more_letter> :: =<letter><more_letter> | _<more_letter> | <digit><more_letter> | <digit> | <letter>
<digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<operation_name> :: = <char_string>
<parameters> :: = <parameter_pair> “,”<parameters> | <parameter_pair>
<parameter_pair> :: = <return_type>< char_string>
<constraint> :: = < pre_condition><post_condition> | <pre_condition> | <post_condition>
<pre_condition> :: = PRE<const_string>
<post_condition> :: = POST<const_string>
<const_string> :: = <char_string> | <special_char><const_string> | <special_char>
<special_char> :: = $\forall$ | $\exists$ | $\wedge$ | $\vee$ | $\neg$ | $\vdash$ | $\exists_1$ | $\emptyset$ | $\in$ | $\notin$ | $\cup$ | $\cap$ | $\Rightarrow$ | $\Leftrightarrow$ | $\neq$ | $\nrightarrow$ | $\twoheadrightarrow$ | $\rightarrowtail$ | $\rightarrow$ | $\mapsto$ | $\rightarrow$ | $\rightarrowtail$ | $\twoheadrightarrow$ | $\rightarrowtail$ | $\twoheadrightarrow$ | λ | μ

EBNF is a grammar used for checking the parsing of operation signature description from UML diagrams. EBNF may be used in deriving parse trees for an automated translator. A developer may have written constraints in first order predicate logics. Here the special_char would allow for the parsing of such constructs; with the assumption that the developer has properly defined the format statement.

3.3 Operation Transformation Rules

In the following subsections transformation for each parts of an operation will be described. The development of the operation signature will be governed by these rules.

3.3.1 Defining Operation Basic Types Schemata

Declare all the necessary data types before schema definitions. Data types in Z are often referred to as basic types or given sets of the specifications. A feature of the Z notation is that it offers a calculus for building large specifications from smaller components [9] – and basic types facilitate this. The importance of basic types and given sets is that they allow one to categorize real world entities into sets. These sets are an essential part of Z schemas because they are used to represent objects and their respective attributes. In this work, basic types will be represented in capitalized letters so that they can be easily identified. The software engineer must examine the attributes of each UML class to identify types that do not have an equivalent representation in Z. Presently, the Z Mathematical Toolkit only directly supports integers [44]. Therefore, other data types needs to be defined. For any string that is not of the type INTEGER ($\mathbb{Z}$), a basic type will be created for it in the Z specification. The process of declaring basic types is not entirely automatable, because some data types will require manual intervention to ensure that they are representative of the parameters. However, the process of extracting the name of the data type and declaring them in the Z specification can be automated. If parameter does not have an associated

data type, and such misrepresentations arise sporadically in UML models, the name of the parameter will be declared and used as a Z basic type [12]. Examples of declaring an operation basic type schemata based on the class diagram and operations {(*convert_heading(heading: double): double, convert_to_internal_speed(speed: double):double and convert_to_external_speed(speed:double): double*} found in Figure 4 is: [DOUBLE].

3.3.2 Define Parameter List Schemata

This step encompasses the description of the Z schema that will contain parameter of each operation. Each UML operation may contain zero or more parameters. Hence, one of two possible options must be taken:

Option 1: UML operation with no parameter

In this option, the definition of a parameter schema is nonessential and any attempt to define a representative Z schema would be illogical and result in rejection by Z/EVES. For example, the *update_display* operation in the *Radar Display* class diagram, Figure 4, contains no parameter. Therefore, no parameter schema definition is necessary.

Option 2: UML operation with one or more parameters

The parameter of each operation will be declared in a parameter type schema. This step is performed successively on each parameter of the UML operation, in two stages, to determine: 1) the name of the parameter and the data type associated with the parameter; and 2) any constraints (values) associated with the parameters. Initially, a one-to-one mapping must be established between parameter /(s) and one of the previously defined basic types or a data type that exist in the Z mathematical toolkit. For the latter phase, parameters along with their respective values will be determined. Constraints that are either domain-specific or operational will be depicted in the schema predicate section. The naming convention used for parameter list schemata is the name of

the parameter followed by the keyword ‘parameter’. For differentiation purpose, each parameter will have an associated index/counter since the same parameter may appear in multiple operations. With reference to Figure 4, the following constraints in Table 3 govern the parameters of each operation. For future work, a format for expressing constraints can be developed, for instance ATT_NAME : <value_range>. An example is given for the *convert_heading* operation in the Aircraft class diagram of Figure 4, which contains one parameter. Their equivalent parameter type schema is:

heading_parameter_01

$$heading : \mathbb{P}DOUBLE$$

$$\forall\, h : heading \bullet 0 \leq h \leq 180$$

3.3.3 Defining Parameter Configuration Schemata

Operations in a class may contain parameters as an item of their execution. This step will be conducted only if an operation accepts parameters. The configuration schema includes all previously defined parameter types. When creating these configuration schemata, each item in the parameter list of an operation is included as the definition of the parameter type. Where each parameter will be identified by its name and corresponding basic type, thus mapping each parameter name to a Z data type or a basic type. These steps should be repeated for each operation that utilizes parameters in their operation implementation. The naming convention used for parameter configuration schemata is the name of the operation followed by the keyword ‘parameter’. Operations are governed by pre-conditions, post-conditions or a combination of both. Where there is no way to automate the pre-condition or post-condition, comments will be utilized. An Example of defining parameter configuration schemata based on the *convert_heading* operation found in Figure 4 Aircraft class is:

Convert_heading_Parameter_01

heading_parameter_01

The Aircraft class operations {*(convert_to_internal_speed (Speed: double) double) and Radar_Display class (move (x: integer, y: integer))*} have three associated parameters of types d*ouble*, and i*nteger*. Integers are present in the Z mathematical toolkit and should not be declared as a basic type. However, the data type called Double are native to some modeling environments but not all; neither is it specified in the Z mathematical toolkit. The basic type parse should therefore identify it as a new basic type. In today's common computer processors, a data type of *Double* precision is essentially a real number with a 64-bit constraint on its size. Manual intervention could change the data types, which were declared as *double* to a basic type called, *real*. Take into consideration that simply changing the name of the basic type from *double* to *real* is semantically equivalent to any proof tool. Therefore, changes of this nature to a basic type will require that constraints be enforced on the data type. Otherwise, the manual intervention would be unproductive.

In the Z specification of the Aircraft Class Diagram, it was important to state what constitutes a *double*. Since real numbers were not originally defined in the Z mathematical toolkit and many existing implementations for real numbers are incorrect [45], there was a need for an appropriate representation for these attributes. Previous work by [45] contains specifications for representing floating point values in Z. However, such effort is outside the scope of this work. In addition, the implementation is very rigid and will not evolve if the size of floating point values increase in future processors.

There are many arguments surrounding the implementation of real numbers and other floating point values. However, a key feature, which separates integers from floats, is that floating point values account for both a numeric precision and a scale whereas integers are whole numbers;

i.e. floating points are approximated values whereas integers are their exact values. The analytical nature of formal methods does not require such distinction to make valid assertions about the system. Therefore, substituting real numbers for integers will suffice.

The result of the basic type parse will return the following: [INTERGER] and [DOUBLE]; manual intervention can substitute DOUBLE with integers – where necessary.

This schema definition incorporates the parameter type schemata for all parameters that exist in an operation.

3.3.4 Define Operation Schemata

After defining parameter configuration schemata, operation schemata is declared. It is mandatory for all methods to have a name. A method that does not have a name will result in compilation error. Making use of schema inclusion, an operation schema is defined by incorporating the associated parameter schema. Additionally, any other variables local to an operation are declared and where necessary constraints on variables or parameter values are defined in the predicate part of the schemata. Operations with the same name may appear in different classes; therefore, a counter/index is utilized to identify each operation. The naming convention used for operation schemata is the name of the operation followed by the keyword 'operation'. Key notational conventions are used in the operation schema definition, which indicates if the execution of a specific operation changes the state of the system. Δ *Aircraft* means that there is a change in the state of the schema after the execution of an operation. See Table 4 that provides a list of notation and their definition that will be utilize in an operation schema. An Example of defining operation schema based on the *convert_heading* operation found in Figure 4 Aircraft class is:

Convert_heading_Operation_01

Δ *Aircraft*

Convert_heading_Parameter_01

heading′ :ℙ *DOUBLE*

---

heading′ = *heading*

3.3.5 Defining Configuration Schema

This schema will incorporate operations to previously define class schemas that were defined by Clachar *et. al*. Updated class schema will include operation schema.

3.4 Transformation Rule Algorithms

Below are algorithms for each transformation rule that were define in section 3.3.1 to 3.3.4

```
begin
for all class in the class diagram
  for all operation in the class
    for each type
     if type! =ℤ
       if type! =blank
        basic_type is USER_DEFINE_TYPE
       else
        basic_type is OPERATION
       endif
     endif
       create basic type schema
 endfor
  endfor;
endfor;
endbegin
```

Figure 6. Algorithm for Defining Operation Basic Type Schemata

Figure 6 illustrates the steps corresponding to defining operation basic type schemata in Z. Each operation must be associated with a basic type in Z, if the basic type is not found in Z then one is define and is refer to as a user define type. Operations that have no associated type are assigned a basic type, that is, the operation name. All basic types are represented in block letter. This process is repeated until all basic types are defined.

```
begin
int count= 0;

for all class in the class diagram
  for all operation in the class
   for each parameter in the operation
     create schema name "parameter name_PARAMETER_[count++]"
     create parameter schema
     if constraints presents
              add constraints
     endif
   endfor
  endfor;
endfor;
endbegin
```

Figure 7. Algorithm for Defining Parameter Schemata

Figure 7 shows the process for defining one or more parameter found within an operation. A counter value is ascribe to a parameter name as an index. This index value diffrentiates each parameter in an operation, since more than one operation within a class may have the same parameter name. Any constraints relating to a paramter are also define in the schema.

```
begin
int count = 0;

  for all class in the class diagram
    for each operation in the class
        if parameter exist {
            create configuration schema name "operation name_PARAMETER_ [count++]"
          schema include all operations parameter schema }
        endif
    endfor;
endfor;
endbegin
```

Figure 8. Algorithm for Defining Parameter Configuration Schemata

To define a parameter configuration schemata, the folowing steps outlined in Figure 8 must be adhered to. The schemata incorporates all previously defined parameter schemata that is associtaed with the operation. An index is also attached to each schema name.

```
begin
int count= 0;

for all class in the class diagram
  for each operation in the class
     create operation schema name "operation name_operation_[count++]"
  if paramater exist
     schema include parameter configuration schema
  endif
  if constrainrs exist
     add operation constraints
   endif

  endfor;
endfor;
endbegin
```

Figure 9. Algorithm for Defining Operation Schemata

Figure 9 depicts the process for defining operation schemata. An operation schema is defined by incorporating the parameter configuration schemata with an index value join to the name of the schema. Any constraint that is placed on the operation is added also.

CHAPTER IV
CASE STUDY
4.1 Description of the Aircraft System

The growing social and economic interest in new unmanned aircraft systems (UASs) applications demands that UASs operate beyond the segregated airspace they are currently able to fly. Unmanned aircraft are not currently permitted access to national air space (NAS) in the United States without special permission from the Federal Aviation Administration (FAA). However, UAS operations in non-segregated airspace should be regulated by aeronautical authorities before UASs can share air space with manned aerial vehicles (MAV). Despite the existence of technologies that could facilitate the integration and operation of UASs in non-segregated airspace, several obstacles remain, mainly UAS safety conditions and airworthiness independent of application. For example, one of the primary concerns with integrating unmanned aircraft is their inability to robustly sense and avoid other aircraft [47]. Another current barrier to the integration of UASs is related to the cultural perception of its risks [48]. According to National Transport Safety Board (NTSB), injury and damage by NTSB classification for U.S. Air carriers operating under 14 CFR 121 for the year 2012 is 16 and 11 respectively, see Table 2 [46]. Table 2 outline accidents with four types of classifications (that is, major, serious, injury and damage) that occurred during 2008 to 2012 irrespective of compliance to aircraft regulations. This shows that while there are regulations that govern air carriers, some form of formalism is required to prevent accidents or catastrophic events. In order for UASs to fly safely into civil airspace, the

development of vigorous testing of UASs, both in laboratory and field experimentation, are key prerequisites.

The United States Air Force Academy (USAFA) is actively involved in unmanned aircraft research across numerous departments involving many projects, aircraft, government agencies, and experimental programs. The importance of these research projects to the Academy, the faculty, the cadets, the Air Force, and to the defense of the nation cannot be understated. In an effort to be proactive in cooperating with recent concerns from the FAA about the growth and proliferation of UAS flights, the Air Force has implemented several new guidelines and requirements. Complying with these guidelines, directives, and regulations has been challenging to researchers and the conduct of research activities at USAFA. Finding ways to incorporate these new guidelines effectively and efficiently is critical to research and participation in joint projects and exercises [49].

To ensure the reliability of these systems, both MAVs and UASs must operate within the same domain that is, US NAS. However, a system must be in place that deals with any form of collision [53]. This aim led to the development of a UAS Research, Development and Design Project at UND[3]. The project goal is to ascertain how practical it is for UASs to operate in an unrestricted airspace, in low-density populated area.

The UND –UAS Research, Development, and Design Project architecture is composed of three main components: a radar system, a data computation unit, and a displays system. The display and data computation system operations is the focus of the work presented here in.

[3] http://www.uasreasearch.com/aboutus/projects.aspx

Table 2. Accident by NTSB Classification, 2008 through 2012 for U.S. Air Carriers Operating Under CFR 121

	Accidents			
Year	Major	Serious	Injury	Damage
2008	4	1	8	15
2009	2	3	15	10
2010	1	0	14	14
2011	0	0	19	12
2012	0	0	16	11

Definition of NTSB Classifications:

- Major - an accident in which any of the three conditions are met:
  A part 121 aircraft was destroyed or there were multiple fatalities or there was one fatality and a part 121 aircraft was substantially damaged.
- Serious - an accident in which at least one of the two conditions are met:
  There was one fatality without substantial damage to a part 121 aircraft or there was at least one serious injury and a part 121 aircraft was substantially damaged.
- Injury - a nonfatal accident with at least one serious injury and without substantial damage to a part 121 aircraft
- Damage - an accident in which no person was killed or seriously injured, but in which any aircraft was substantially damaged

4.2 Application of Methodology

In this section, the transformations rules that were developed in Chapter III will be applied to a subset of the UAS system. The rules will apply to all operations of Figure 10, for the complete transformation of the case study please see Appendix A:

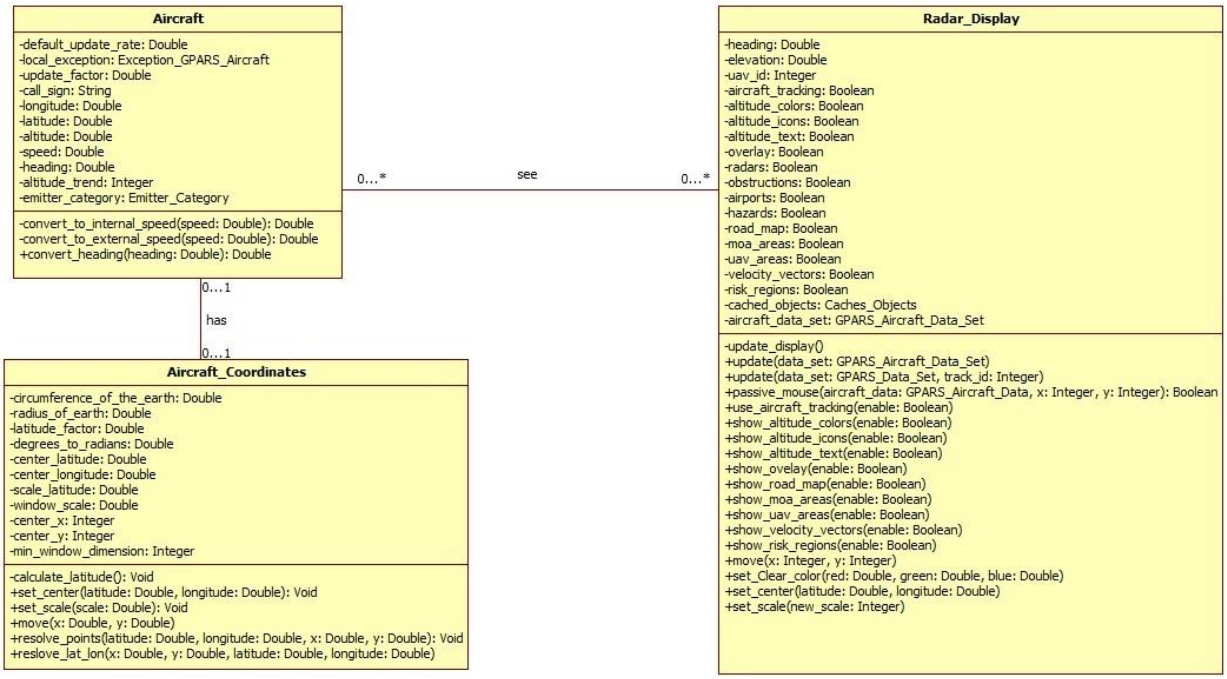

Figure 10.UAS Aircraft and Radar Class Diagram

The above diagram, Figure 10 is a small subset of classes from the system model currently being formalized. This will be used to demonstrate the execution of the transformation rule on the class operations. These transformation rules include:

1) Step 1: Defining basic types
2) Step 2: Defining Parameter Schemata
3) Step 3: Defining Parameter Configuration Schemata
4) Step 4: Defining Operation Schemata

4.2.1 Constraints on Class Diagrams

Table 3 represents constraints that govern each class diagram in Figure 10. As a final step in the transformation rule, constraints are manually included in the schema of the operation.

Table 3. Constraints for Aircraft, Radar_Display and Aircraft_Coordinates Class Operation Attributes

Constraints on Aircraft Class Attributes The following constraints govern the Aircraft class:-	
Speed:	All speed have a lower and upper bound. The speed of the aircraft should not exceed the speed of supersonic. The minimum and maximum speed for the Aircraft are: - min_speed = 0.0 and max_speed = 250 knots
Heading:	The minimum heading = 0.0; maximum = 360.0
Constraints on Radar_Display Class Attributes:	
passive_mouse:	The boundary for the mouse drawing on the x axis is: 0.00 to 180.00 degrees The boundary for the mouse drawing on the y axis = 0.00 to 180.00 degrees
set_center:	The center can begin from anywhere between -90.0 to 90.0 in latitude (across) and -180.0 to 180 longitude (down).
set_scale:	1 inches = 100 foot
Constraints on Aircraft_Coordinates Class Attributes:	
Latitude :	Minimum latitude = -90.0; maximum latitude = 90.0

Longitude:	minimum longitude = -180.0; maximum longitude = 180.0
Altitude:	minimum altitude = -3000.0; maximum altitude = 168960.0

An important feature of formal specification is that of "state". A system can be in one of several different states. Z captures a system change of state base on the data that a system store and how data are change in the schemas. Some of the notations used in this research are listed below with their corresponding uses.

Table 4. Notations Used to Specify Change in the State of Operation Schemas and their description

Notation	Symbol	Example	Description
Delta	Δ	Δ *Aircraft*	Shows that there is a change in the state of the schema after the execution of an operation
Xi	Ξ	Ξ *Radar_Display*	Demonstrates that there is NO change in the state of the schema after the execution of an operation
Prime variables	$'$	X'	Conventionally used to represent the value of a variable after an operation
Unprimed variables		X	Value of a variable before execution of an operation
For all	$\forall$	$\forall x : xed \bullet 0 \geq x \leq 10$	Forall x:X | P1 • P2 means: any element of X that satisfies P1 also satisfies P2

Table 5 illustrates a subset of the operation signature schemas that were developed from conducting the formalization techniques outlined in chapter III, on the class diagram of Figure 10. Appendix A has the completed schema for Figure 10.

In the example that follows three (3) basic types were defined, specifically, [DOUBLE], [GPARS_AIRCRAFT_ DATA], and [BOOLEAN], five (5) parameter schemata, three (3) configuration parameter schemata and four (4) operation schemata. The parameter schemas are *convert_heading_Paramaters*, *convert_to_internal_speed_Paramaters* and

passive_mouse_Parameters. The operation schemas are *convert_heading*, *convert_to_internal_speed*, *update_display*, and *passive_mouse.*

Table 5. Z Schemas for the UML Class Diagram of Figure 10

Operation Name: *Convert_heading(heading: Double): Double*

Step 1: [DOUBLE]

Step 2: *heading_parameter_01*

$$\begin{array}{|l}
\hline
heading : \mathbb{P}DOUBLE \\
\hline
\forall\, h : heading \bullet 0 \leq h \leq 180 \\
\hline
\end{array}$$

Step 3: *Convert_heading_Parameter_01*

$$\begin{array}{|l}
\hline
heading_parameter_01 \\
\hline
\end{array}$$

Step 4: *Convert_heading_Operation_01*

$$\begin{array}{|l}
\hline
\Xi\ Radar_Display \\
\Xi\ Aircraft_Coordinates \\
\Delta\ Aircraft \\
Convert_heading_Parameter_01 \\
heading' : \mathbb{P}DOUBLE \\
\hline
heading' = heading \\
\hline
\end{array}$$

Table 5. cont.

Operation Name: *Convert_to_internal_speed(Speed: Double): Double*

Step 1: [DOUBLE]

Step 2: *_speed_paramater_02*

$Speed : \mathbb{P} DOUBLE$

$\forall\ s : speed \bullet 0.0 \leq s \leq 250\ knots$

Step 3: *_Convert_to_internal_speed_Paramater_02*

speed_parameter_02

Step 4: *_Convert_to_internal_speed_Operation_02*

$\Xi\ Radar\_Display$

$\Xi\ Aircraft\_Coordinates$

$\Delta\ Aircraft$

Convert_to_internal_Speed_Parameter_02

$speed' : \mathbb{P} DOUBLE$

$speed' = speed$

Operation Name: *Update_display ()*

Step 1: [UPDATE_DISPLAY]

Step 2 and 3 are ignored because there is no parameter for *update_display* operation

Step 4: *_Update_display_03*

$\Xi\ Aircraft$

Table 5. cont.

Operation Name: *passive_mouse(Aircraft_data:GPARS_Aircraft_DATA, x: Integer, y: Integer): Boolean*

Step 1: [GPARS_AIRCRAFT_DATA]
[BOOLEAN]

Step 2: *_Aircraft_data_parameter_03_*

$Aircraft\_data : GPARS\_AIRCRAFT\_DATA$

_x_parameter_04

$x : \mathbb{P}\mathbb{Z}$

$\forall X : x \bullet 0.00 \leq X \leq 180.00$

_y_paramater_05

$y : \mathbb{P}\mathbb{Z}$

$\forall Y : y \bullet 0.00 \leq Y \leq 180.00$

Step 3: *_passive_mouse_Paramater_03*

air_craft_parameter_03
x_paramterer_04
y_parameter_05

Step 4: *_passive_mouse_Operation_04*

$\Xi\ Aircraft$
$\Delta\ Radar\_Display$
passive_mouse_Parameter_03
$Aircraft\_data' : GPARS\_AIRCRAFT\_DATA$
$x' : \mathbb{P}\mathbb{Z}$
$y' : \mathbb{P}\mathbb{Z}$

$Aircraft\_data' = Aircraft\_data$
$x' = x$
$y' = y$

A basic type and parameter type schema will not be defined for the *update_display* operation. The reason seeing that, this operation signature carries no data type and no parameter. According to the define parameter list schemata rule, this step will be conducted only if an operation accepts parameters.

4.3 Results and Analysis

In the methodology above, formal methods were applied on operations to demonstrate the application of the refinement and transformation process. The component of subset class diagram obtained from the UAS Risk Mitigation system contained 3 classes, and 31 operations (27 user defined basic type, 44 parameter schemata, 29 configuration parameter schemata, and 31 operation schemata. Operations also consist of constraints for some parameters in operations. The application of the process defined in the methodology provides a realistic way of applying formal methods rather than theoretical considerations. Still, the work that was involved in carrying out this project was very tedious which introduced periodic errors. Thus, implementing a tool to automate the formalization process would be beneficial. This would simplify the conversion (Schema definitions), reduce the workload, and lessen the probability of human errors in the specification.

CHAPTER V
CONCLUSION

This research defines and illustrates the steps involved in deriving operation schema for UML class diagrams of a safety critical system. In many software applications such as in the safety critical areas it is important to have correct and bug free software. Formal specification is one such approach to produce good quality, correct and error free software. The purpose of using notation like Z is to produce an accurate specification from initial client requirements. The notation has a restricted syntax so it is precise but still abstract enough so as not to constrain how a developer will go on to design application. This study supports the need for reliable development methodologies for safety critical systems and for avionic system development to comply with industry standard, DO-178C specification. It is an extension of previous work done by Clachar and Jackson that concentrated on formalizing, and verifying and validating UML software models for safety critical systems [1] [2].

One of the principal concerns with amalgamating unmanned aircraft into national air space is their lack of ability to robustly sense and avoid other aircraft. Systems such as these must adhere to industry standard, for instance RTCA-DO178B, because they are classified as been safety critical. To ensure that catastrophic events (for example, loss of life) do not occur, accuracy in safety critical systems is necessary.

Unified Modeling Language is the ISO standard for modeling systems. The class diagram is one type of UML model used to express systems requirements of stakeholders and to discover

additional systems requirements. However, UML lacks precision when expressing design decisions. Textual descriptions are used to express characteristics of the system, which cannot be captured by UML. This further introduces another level of ambiguity in the models – since they are usually expressed in natural language. Hence, the need for a meta- model (EBNF) that would bring more formatting and understanding to the work conducted in this research. One method that is used to remove ambiguity in models is to transform UML models into an analyzable representation using formal specification techniques (FSTs). FSTs are based on mathematical logics, which makes use of first order logics and set notation. Adopting such approach to system development plays an important role in safety critical system.

FSTs have been in existence prior to the beginning of UML. However, unlike UML it does not have a high level of simplicity that makes its models easily communicated to stakeholders. Currently, the formalization process is conducted manually. To make research on FSTs more worthy, some degree of automation is imperative. Therefore, conducting a case study in the area of automated tools for FSTs in safety critical systems will be beneficial in enlightening researchers on the complexity, advantages, and possible use of such software.

This case study supports research that identify the benefits of the application of formal methods to industries such as formal specification of an oscilloscope (Tektronix) and formal methods in safety-critical railway systems. In the former study, the researcher adopted formal methods to gain insight into system architecture. In the latter work, the B formal method was used in the development of platform screen door controllers. Both investigations concluded that the application of formal specification appears to be precise, efficient, and well suited to address projects requiring high level of safety [59] [60].

Besides applying this methodology to UASs, the contribution of this research may be extended to automotive control systems (for example, factory, marine, space exploration, robotics, and other specialist areas) where formalism is a necessity. The use of formal methods is an effective mean to improve complex systems reliability and quality. Benzadri *et. al.* adopted a formal method that utilized modeling interactions between cloud services and customers. The researchers combined Cloud customers' bigraph and Cloud services bigraph to formally specify Cloud services structure. This study is applicable to formalizing Cloud computing concepts and to overcome one of cloud computing main obstacles, specifically bugs in large scale Distributed Systems – "one of the difficult issues in cloud computing is removing errors in these very large scale distributes systems" [57] [58]. The main issue that still needs to be addressed is the crucial absence of an appropriate model for cloud computing. This research may possibly be able to support major Cloud computing concepts specification and allow formal modeling of high-level services provided over Cloud computing architecture.

5.1 Future Work

The methodology was successfully applied to the operation of the UAS system. Other efforts can apply the same methodology to other systems to prove the validity or accuracy of the methodology. In addition, since the focus of this work was on refining transformation rules from an informally defined design in UML to one that is verifiable, formal specification; subsequent efforts can derive a process for expressing constraints. An attempt that can be made is to develop a command-line toolkit to automate the steps outlined in the methodology for both the EBNF and the transformation rules. Such tool would accept an operation and ensure the format abide by the EBNF configuration. Subsequently, the tool would apply the refinement steps to each operation by decomposing operation into small pieces (Schemas). The automated toolkit can then be added on

as an additional feature of Z/EVES to demonstrate the consistency of refinement and to identify errors.

APPENDIX

Z Schemas

[DOUBLE]

[GPARS_AIRCRAFT_DATA]

[BOOLEAN]

[UPDATE_DISPLAY]

[VOID]

[MOVE]

[RESOLVE_LAT_LON]

[UPDATE]

[GPARS_AIRCRAFT_DATA_SET]

[USE_AIRCRAFT]

[SHOW_ALTITUDE_COLORS]

[SHOW_ALTITUDE_ICON]

[SHOW_ALTITUDE_TEXT]

[SHOW_OVERLAY]

[SHOW_RADARS]

[SHOW_OBSTRUCTION]

[SHOW_AIRPORTS]

[SHOW_HAZARDS]

[SHOW_ROAD_MAP]

[SHOW_MOA_AREAS]

[SHOW_UAV_AREAS]

[SHOW_VELOCITY_VECTORS]

[SHOW_RISK_REGIONS]

[MOVE]

[SET_CLEAR_COLOR]

[SET_CENTER]

[SET_SCALE]

speed_paramater_01

speed: ℙ*DOUBLE*

∀ *s*: *speed* • 0.0 ≤ *s* ≤250 *knots*

Convert_to_internal_speed_Paramater_01

speed_parameter_01

Convert_to_internal_speed_Operation_01

Ξ *Radar_Display*
Ξ *Aircraft_Coordinates*
Δ *Aircraft*
Convert_to_internal_Speed_Parameter_01
speed ′: ℙ*DOUBLE*

---

speed ′ = speed

speed_parameter_02

speed: ℙ*DOUBLE*

∀ *s*: *speed* • 0.0 ≤ *s* ≤ 250 *knots*

Convert_to_external_speed_Parameter_02

speed_paramater_02

Convert_to_externa_speed_Operation_02

Ξ *Radar_Display*
Ξ*Aircraft_Coordinates*
Δ*Aircraft*
Convert_to_external_Speed_parameter_02
speed′: ℙ*DOUBLE*

---

speed′ = speed

heading_parameter_03

heading: ℙ*DOUBLE*

$\forall\ h : heading \bullet 0 \leq h \leq 180$

Convert_heading_Parameter_03

heading_parameter_03

Convert_heading_Operation_03

Ξ *Radar_Display*

Ξ *Aircraft_Coordinates*

Δ *Aircraft*

Convert_heading_Parameter_03

heading′ : ℙ*DOUBLE*

---

heading′ = *heading*

Calculate_latitude_Operation_04

Ξ *Aircraft*

Latitude_parameter_04

Latitude: ℙ*DOUBLE*

$\forall\ lat: latitude \bullet -90.0 \leq lat \leq 90.0$

Longitude_parameter_05

Longitude: ℙ*DOUBLE*

$\forall\ lon: longitude \bullet -180.0 \leq lon \leq 180.0$

Set_center_Parameter_04

Latitude_parameter_04

Longitude_paramater_05

Set_center_Operation_05

Ξ *Aircraft*

Δ *Aircraft_Coordinates*

Set_Center_Parameter_04

Latitude′ : $\mathbb{P}$*DOUBLE*

Longitude′ : $\mathbb{P}$*DOUBLE*

Latitude′ = *latitude*

Longitude′ = *longitude*

Scale_parameter_06

Scale: $\mathbb{P}$*DOUBLE*

$\forall$ *s*: *scale* • 1 *inches* = 1 *foot*

Set_scale_parameter_05

Scale_paramater_06

Set_Scale_Operation_06

Ξ *Aircraft*

Δ *Aircraft_Coordinates*

Set_scale_parameter_05

Scale′ : $\mathbb{P}$*DOUBLE*

Scale′ = *scale*

x_parameter_07

x: $\mathbb{P}$*DOUBLE*

---

y: *parameter_08*

y: $\mathbb{P}$*DOUBLE*

move_paramater_06

x_parameter_07

y_parameter_08

move_Operation_07

Ξ *Aircraft*

Δ *Aircraft_Coordinates*

Move_parameter_06

$X' : \mathbb{P}DOUBLE$

$Y' : \mathbb{P}DOUBLE$

$X' = x$

$Y' = y$

Latitude_parameter_09

Latitude: $\mathbb{P}DOUBLE$

$\forall$ *lat*: *latitude* $\bullet$ $-90.0 \leq lat \leq 90.0$

Longitude_parameter_10

Longitude: $\mathbb{P}DOUBLE$

$\forall$ *lon*: *longitude* $\bullet$ $-180.0 \leq lon \leq 180.0$

x_parameter_11

$x : \mathbb{P}DOUBLE$

---

y_parameter_12

y: $\mathbb{P}DOUBLE$

---

resolve_points_Parameter_07

latitude_parameter_09

longtitude_parameter_10

x_parameter_11

y_parameter_12

resolve_points_Operation_08

Ξ *Aircraft*

Δ *Aircraft_Coordinates*

Resolve_points_Parameter_07

Latitude′ : ℙ*DOUBLE*

Longitude′ : ℙ*DOUBLE*

X′ : ℙ*DOUBLE*

Y′ : ℙ*DOUBLE*

Latitude′ = *latitude*

Longitude′ = *longitude*

X′ = *x*

Y′ = *y*

x_paramater_13

x: ℙ*DOUBLE*

---

y_parameter_14

y: ℙ*DOUBLE*

---

latitude_parameter_15

latitude: ℙ*DOUBLE*

$\forall$ *lat*: *latitude* • $-90.0 \leq lat \leq 90.0$

Longitude_parameter_16

Longitude: ℙ*DOUBLE*

$\forall$ *lon*: *longitude* • $-180.0 \leq lon \leq 180.0$

Resolve_lat_lon_Parameter_08

x_parameter_13

y_parameter_14

latitude_parameter_15

longitude_parameter_16

resolve_lat_lon_Operation_09

Ξ *Aircraft*

Δ *Aircraft_Coordinates*

Resolve_lat_lon_Parameter_08

$X' : \mathbb{P}DOUBLE$

$Y' : \mathbb{P}DOUBLE$

$Latitude' : \mathbb{P}DOUBLE$

$Longitude' : \mathbb{P}DOUBLE$

$X' = x$

$Y' = y$

$Latitude' = latitude$

$Longitude' =: longitude$

Update_display_Operation_10

Ξ *Aircraft*

Data_set_parameter_17

Data_set: *GPARS_AIRCRAFT_DATA_SET*

Update_Parameter_09

Data_set_parameter_17

Update_Operation_11

Ξ *Aircraft*

Δ *Radar_Display*

Update_Parameter_09

$DATA\_SET' : \mathbb{P}DOUBLE$

$DATA\_SET' = data\_Set$

Data_set_parameter_18

Data_set: *GPARS_AIRCRAFT_DATA_SET*

Track_is_parameter_19

Track_is: $\mathbb{P}\mathbb{Z}$

Update_Parameter_10

Data_set_parameter_18

Track_is_parameter_19

Update_Operation_12

Ξ *Aircraft*

Δ *Radar_Display*

Update_Parameter_10

Data_set′ : *GPARS_AIRCRAFT_DATA_SET*

Track_is′ : $\mathbb{P}\mathbb{Z}$

Data_set′ = *data_set*

Track_is′ = *track_is*

Aircraft_data_parameter_20

Aircraft_data: *GPARS_AIRCRAFT_DATA*

x_parameter_21

$x : \mathbb{P}\mathbb{Z}$

$\forall X : x \bullet 0.00 \leq X \leq 180.00$

y_paramater_22

$y : \mathbb{P}\mathbb{Z}$

$\forall Y : y \bullet 0.00 \leq Y \leq 180.00$

passive_mouse_Paramater_11

air_craft_parameter_20
x_parameter_21
y_parameter_22

passive_mouse_Operation_13

Ξ *Aircraft*
Δ *Radar_Display*
passive_mouse_Parameter_11
$Aircraft\_data' : GPARS\_AIRCRAFT\_DATA$
$x' : \mathbb{P}\mathbb{Z}$
$y' : \mathbb{P}\mathbb{Z}$

$Aircraft\_data' = Aircraft\_data$
$x' = x$
$y' = y$

enable_parameter_23

$enable : \mathbb{P}BOOLEAN$

---

use_aircraft_Paramater_12

enable_parameter_23

use_aircraft_Operation_14

Ξ *Aircraft*
Use_aircraft_parameter_12
$enable' : \mathbb{P}BOOLEAN$

$enable' = enable$

enable_parameter_24

$enable : \mathbb{P} BOOLEAN$

show_altitude_colors_Paramater_13

enable_parameter_24

show_altitude_colors_Operation_15

$\Xi\ Aircraft$

$Show\_altitude\_colors\_parameter\_13$

$enable' : \mathbb{P} BOOLEAN$

$enable' = enable$

enable_parameter_25

$enable : \mathbb{P} BOOLEAN$

show_altitude_icon_Paramater_14

enable_parameter_25

show_altitude_icon_Operation_16

$\Xi\ Aircraft$

$Show\_altitude\_icon\_parameter\_14$

$enable' : \mathbb{P} BOOLEAN$

$enable' = enable$

enable_parameter_26

$enable : \mathbb{P} BOOLEAN$

---

show_altitude_text_Paramater_15

enable_parameter_26

show_altitude_text_Operation_17

$\Xi\ Aircraft$

$Show\_altitude\_text\_parameter\_15$

$enable' : \mathbb{P} BOOLEAN$

$enable' = enable$

enable_parameter_27

$enable : \mathbb{P} BOOLEAN$

---

show_overlay_Paramater_16

enable_parameter

show_overlay_Operation_18

$\Xi\ Aircraft$

$show\_overlay\_parameter\_16$

$enable' : \mathbb{P} BOOLEAN$

$enable' = enable$

enable_parameter_28

$enable : \mathbb{P} BOOLEAN$

---

show_radars_Paramater_17

enable_parameter_28

show_radars_Operation_19

Ξ *Aircraft*

Show_radars_parameter_17

enable′ : $\mathbb{P}$*BOOLEAN*

---

enable′ = *enable*

enable_parameter_29

enable: $\mathbb{P}$*BOOLEAN*

---

show_obstruction_Paramater_18

enable_parameter_29

show_obstruction_Operation_20

Ξ *Aircraft*

Show_obstruction_parameter_18

enable′ : $\mathbb{P}$*BOOLEAN*

---

enable′ = *enable*

enable_parameter_30

enable: $\mathbb{P}$*BOOLEAN*

---

show_airports_Paramater_19

enable_parameter_30

show_airports_Operation_21

Ξ *Aircraft*

Show_airports_parameter_19

enable′ : $\mathbb{P}$*BOOLEAN*

---

enable′ = *enable*

enable_parameter_31

$enable : \mathbb{P} BOOLEAN$

---

show_hazards_Paramater_20

enable_parameter_31

show_hazards_Operation_22

$\Xi\ Aircraft$

$Show\_hazards\_parameter\_20$

$enable' : \mathbb{P} BOOLEAN$

$enable' = enable$

enable_parameter_32

$enable : \mathbb{P} BOOLEAN$

---

show_road_map_Paramater_21

enable_parameter_32

show_road_map_Operation_23

$\Xi\ Aircraft$

$Show\_road\_map\_parameter\_21$

$enable' : \mathbb{P} BOOLEAN$

$enable' = enable$

enable_parameter_33

$enable : \mathbb{P} BOOLEAN$

---

show_moa_areas_Paramater_22

enable_parameter_33

show_moa_areas_Operation_24

Ξ *Aircraft*
Show_moa_areas_parameter_22
enable′ : $\mathbb{P}$*BOOLEAN*

---

enable′ = *enable*

enable_parameter_34

enable: $\mathbb{P}$*BOOLEAN*

---

show_uav_areas_Paramater_23

enable_parameter_34

show_uav_areas_Operation_25

Ξ *Aircraft*
Show_uav_areas_parameter_23
enable′ : $\mathbb{P}$*BOOLEAN*

---

enable′ = *enable*

enable_parameter_35

enable: $\mathbb{P}$*BOOLEAN*

---

show_ velocity_vector_Paramater_24

enable_parameter_35

show_ velocity_vector_Operation_26

Ξ *Aircraft*
Show_velocity_vector_parameter_24
enable′ : $\mathbb{P}$*BOOLEAN*

---

enable′ = *enable*

enable_parameter_36

$enable : \mathbb{P}BOOLEAN$

show_risk_regions_Paramater_25

enable_parameter_36

show_risk_regions_Operation_27

$\Xi\ Aircraft$

Show_risk_regiond_parameter_25

$enable' : \mathbb{P}BOOLEAN$

$enable' = enable$

x_parameter_37

$x : \mathbb{P}\mathbb{Z}$

y_parameter_38

$y : \mathbb{P}\mathbb{Z}$

move_Paramater_26

x_parameter_37

y_parameter_38

move_Operation_28

$\Xi\ Aircraft$

$\Delta\ Radar\_Display$

move_parameter_26

$x' : \mathbb{P}\mathbb{Z}$

$y' : \mathbb{P}\mathbb{Z}$

$x' = x$

$y' = y$

red_parameter_39

$red : \mathbb{P} DOUBLE$

green_parameter_40

$green : \mathbb{P} DOUBLE$

blue_parameter_41

$blue : \mathbb{P} DOUBLE$

set_clear_color_parameter_27

$red\_parameter\_39$
$green\_parameter\_40$
$blue\_parameter\_41$

set_clear_color_Operation_29

$\Xi\ Aircraft$
$\Delta\ Radar\_Display$
$Set\_clear\_color\_parameter\_27$
$Red' = \mathbb{P} DOUBLE$
$Green' = \mathbb{P} DOUBLE$
$Blue' = \mathbb{P} DOUBLE$

$Red' = red$
$Green' = green$
$Blue' = blue$

Latitude_parameter_42

$Latitude : \mathbb{P} DOUBLE$

$\forall\ lat : latitude \bullet -90.0 \leq lat \leq 90.0$

Longitude_parameter_43

$Longitude : \mathbb{P} DOUBLE$

$\forall\ lon : longitude \bullet -180.0 \leq lon \leq 180.0$

Set_center_parameter_28

Latitude_parameter_42

Longitude_parameter_43

Set_center_Operation_30

Ξ *Aircraft*

Δ *Radar_Display*

Set_center_parameter_28

Latitude′ = $\mathbb{P}$*DOUBLE*

Longitude′ = $\mathbb{P}$*DOUBLE*

Latitude′ = *latitude*

Longtitude′ = *longitude*

New_scale_parameter_44

New_scale: $\mathbb{P}\mathbb{Z}$

Set_scale_parameter_29

New_scale_paramater_44

Set_scale_operation_31

Ξ *Aircraft*

Δ *Radar_Display*

Set_scale_parameter_29

New_Scale′ : $\mathbb{P}\mathbb{Z}$

New_scale′ = *new_scale*

REFERENCES

[1] Clachar, S., Grant E. A Case Study in Formalizing UML Software Models of safety Critical Systems. In Proceedings of the Annual International Conference on Software Engineering. Phuket, Thailand (2010)

[2] Jackson, V., Grant, E. Verification & Validation of Object-Oriented Functional Design using Specification Techniques, In Proceedings of the 44th Annual Midwest Instruction and Computing Symposium, Duluth, MN (2011)

[3] U.S. Dept. of Defense: FY2009-2034: Unmanned Systems Integrated Roadmap, 2009.

[4] Sazdovski,V., Kolemishevska-Gugulovska, T., Stankovski, M.:Kalman Filter Implementation for Unmanned Aerial Vehicles Navigation Developed with a Graduate Course. Institute of ASE at Faculty of EE (2005) St. Cyril and Methodius University, MK-1000, Skopje, Republic of Macedonia

[5] RTCA, Inc, EUROCAE: DO-178B, Software Considerations in Airborne Systems and Equipment. SC-167 (1992) RTCA, Washington DC, USA

[6] Brosgol M. B.: Safety and security: Certification issues and Technologies. CrossTalk: The Journal of Defense and Software Engineering vol. 21 (10) 9--14 (2008).

[7] ISO/IEC 19501, Information Technology - Open Distributed Processing,: Unified Modeling Language (UML) Version 1.4.2 (2005)

[8] France, R. B., Evans, A., Lano, K., Rumpe, B.: The UML as a Formal Modeling Notation. In Computer Standards & Interfaces, vol 19, issue 7, 325--334 (1998)

[9] Hall, A.: Using Z as a Specification Calculus for Object-Oriented Systems. In Proceedings of the Third International Symposium of VDM Europe on VDM and Z - Formal Methods in Software Development, 290--318 (1990)

[10] ISO/IEC 13568, Information Technology: Z Formal Specification Notation - Syntax, Type System and Semantics. First ed. ISO/IEC 2002)

[11] Hall, A.: Seven myths of formal methods, Software, IEEE , vol.7, no.5, 11--19, (1990)

[12] France, R.B., Bruel, J., Larrondo-Petrie, M.M.: An Integrated Object-Oriented and Formal Modeling Environment. In Proceedings of JOOP. 25--34. (1997)

[13] Bowen, J., (2003). Formal Specification and Documentation using Z: A Case Study Approach. Revised 2003. pp 4-6.

[14] Pressman, R., (2005). *Software Engineering: A Practitioner's Approach*. Boston, Mass.: McGraw-Hill.

[15] Snook, C. & Butler, M. (2006). *UML-B: Formal Modeling and Design aided by UML*. ACM Transactions on Software Engineering and Methodology (TOSEM) Volume 15 Issue 1, New York, NY, USA.

[16] Shroff, M., France, R. B., (1997). *Towards a Formalization of UML Class Structures in Z.* COMPSAC '97 - 21st International Computer Software and Applications Conference. pp.646

[17] Naur, P. ed.: Revised Report on the Algorithmic Language Algol 60, Communications of the ACM,6(1), 1-17,1963

[18] Naur, P.: The European Side of the Last Phase of the Development of Algol,ACM SIGPLAN Notices,13,15-44, 1978

[19] Johnson, S.C.: Yacc - Yet Another Compiler Compiler, AT&T Bell Laboratories, Computing Science Technical Report No.32, AT&T Bell Labs. Murray Hill, NJ, 1975.

[20] Lesk, M.E.: Lex - A Lexical Analyzer Generator, AT&T Bell Laboratories, Computing Science Technical Report No.39, Murray Hill, NJ, 1975

[21] Parr, T.:ANTLR Reference Manual, http://www.antlr.org/, 2000

[22] Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation, Electronic Notes in Theoretical Computer Science, vol 152, In Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), 125-142, ISSN 1571-0661 (2006)

[23] Chikofsky, E. J., Cross II., J. H.: Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software. *Vol.* 7, 1, 13—17 (1990).

[24] France, R.B., Bruel, J., Larrondo-Petrie, M.M.: An Integrated Object-Oriented and Formal Modeling Environment. In proceedings of JOOP. 25--34. (1997)

[25] Potter, B., Sinclair J.: An Introduction to Formal Specification and Z. 2nd ed. Prentice Hall (1996)

[26] Hai, H., Yi-fang, Z., Chi-lan, C. "Unified Modeling of Complex Real-Time Control Systems." in *Proc Design, Automation, and Test in Europe. IEEE Computer Society*, - Volume 1, pages 498-499, March 2005.

[27] Bell, D. “UML Basics: The Class Diagram.” Internet: http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/ Sept. 15, 2004 [April, 2009]

[28] Sommerville, I, *Software Engineering 9th Ed.* Addison Wesley, Boston, Massachusetts, 2010.

[29] Dewar, R.B.K. “Integrating Formal Methods into Software Toolsets for Avionics Certification.” Internet: http://www.defensetechbriefs.com/component/content/article/8238 August, 2010 [February 2011].

[30] Berkenkotter, K., “Using UML 2.0 in Real-Time Development: A Critical Review” in *Proc SVERTS Workshop*, 2003.

[31] Saaltink, M. “The Z/EVES 2.0 User’s Guide.” Technical Report TR-99-5493-06a, ORA Canada, One Nicholas Street, Suite 1208 - Ottawa, Ontario K1N 7B7 - CANADA, Oct. 1999.

[32] Sendall, S.; Kozaczynski, W., “Model Ttransformation: The Heart and Soul of Model-Driven Software Development Software”, *IEEE* , vol.20, no.5, pp. 42-45, Sept.-Oct. 2003

[33] Poole, J. D. “Model-Driven Architecture: Vision, Standards And Emerging Technologies.” In *Workshop on Metamodeling and Adaptive Object Models, ECOOP 2001*, 2001.

[34] Malik, P., Utting, M. “CZT: A framework for Z tools”. *In: Treharne*, H., King, S., Henson, M.C., Schneider, S.A. (eds), ZB, LNCS, vol. 3455, pp. 65–84. Springer, Heidelberg 2005.

[35] S. Dupuy, Y. Ledru and M. Chabre-Peccoud, “An Overview of RoZ: A Tool for Integrating UML and Z Specifications,” *Proc. Advanced Information Systems Eng. Conf. (CAiSE"00),* B. Wangler and L. Bergman, eds., pp. 417-430, 2000.

[36] Dupuy, S. “RoZ version 0.3 an Environment for the Integration of UML and Z.” technical report Laboratoire LSR-IMAG, 1999. http://www-lsr.imag.fr/Les.Groupes/PFL/RoZ/index.html

[37] Feynman, R. “EBNF: A Notation to describe Syntax” , pp 1 – 6. http://www.ics.uci.edu/~pattis/ICS-33/lectures/ebnf.pdf

[38] ISO/IEC, EXTENDED BNF 1996.www.dataip.co.uk/Reference/EBNF.php

[39] Xia, Y., Glinz, M. Rigorous EBNF-based Definition for a graphic Modeling Language, Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC’03),IEEE, 2003

[40] Berardi, D., Calvanese, D., Giacomo, G.D. Reasoning on UML Class Diagrams. *Published in journal of Artificial Intelligence, Volume 168 Issue 1-2, October, 2005 , Pages 70-118*

[41] Fowler, M., Scott, K. UML Distilled—Applying the Standard Object Modeling Language, Addison-Wesley,Reading, MA, 1997

[42] Evans, A.S. Reasoning with UML class diagrams, in: Second IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT'98), IEEE Computer Society Press, 1998

[44] Spivey, J.M. The Z Notation: A reference manual. Prentice Hall International, 1998.

[45] Barrett, G., "Formal Methods Applied to a Floating-Point Number System," *Software Engineering, IEEE Transactions*, vol.15, no.5, pp.611-621, May 1989 doi: 10.1109/32.24710

[46] National Transport Safety Board (2012). 2012 Aviation Statistics.

[47] Temizer, S., Kochenderfer, J. M., Kaelbling, P. L., Lozano-Perez, T., and Kuchar, K. J. Collision Avoidance for Unmanned Aircraft using Markov Decision Processes. *American Institute of Aeronautics and Astronautics.*

[48] Gimenes, A.V.R., Vismari, F.L., Avelino, F.V., Camargo Jr, B.J., Almeida Jr, R.J., Cugnasca, S.P. Guidelines for the Integration of Autonomous UAS into the Global ATM. *J Intell Robot Syst (2014) 74:465-478.*

[49] Bushey, E.D. Unmanned Aircraft Flight and Research at the United States Air Force Academy. *J Intell Robot Syst (2009) 54:79-85.*

[50] Hempe, D.W., 20-115C - Airborne Software Assurance. http://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentID/1021710 , July 19, 2013

[51] Ledru, Y. Identifying pre-conditions with the Z/EVES theorem prover. *In Proceeding of the 13th International Conference on Automated Software Engineering.* Pp. 32-41, IEEE Computer Society Press, Honolulu 1998

[52] Saaltink, M. The Z/EVES 2.0 User's Guide. October 1999

[53] Magner, B. "Worries about Mid-Air Collision keep Civilian Drones Grounded" Internet: http://www.nationaldefensemagazine.org/archive/2008/May/Pages/Worries2270.aspx May, 2008 [April 2015]

[54] Federal Aviation Administration. " Fact Sheet – Unmanned Aircraft Systems (UAS) Internet: https://www.faa.gov/news/fact_sheets/news_story.cfm?newsId=14153 January 2014 [April 2015]

[55] ISO/IEC 14977:1996 International Standard. Information Technology – Syntactic metalanguage – Extended BNF

[56] Jones, D.M. "The New C Standard: An Economic and Cultural Commentary". January 2008

[57] Armbrust, M., Fox, A., Grith, R., Joseph, A.D., Katz., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley (2009)

[58] Benzadri, Z., Belala, F., Bouanaka, C.: Towards a Formal Model for Cloud Computing. Service-Oriented Computing – ICSOC 2013 Workshops. Volume 83777, 2014, pp 381-393

[59] Delisie, N., Garlan, D.: A Formal Specification of an Oscilloscope. IEE Software, Volume 7, Number 5, September 1990

[60] Leconte, T., Servat, T., Pouzancre, G.: Formal Methods in Safety-Critical Railway Systems. ClearSy, Aix en Provence, France.

[61] Gupta, S.G., Ghonge, M.M., Jawandhiya, P.M.: Review of Unmanned Aircraft Systems (UAS). *International Journal of Advanced Research in Computer Science Engineering & Technology (IJARCET),* Volume 2, Issue 4, April 2013.

[62] Gogolla, M., Buttner, F., Richters, M. USE: A UML-based specification environment for validating UM and OCL. Science of Computer Programming 69 (2007) 27 - 34